CHAPTER 4

The Engine: Safeguarding Itself before Safeguarding Others

To be a blacksmith, you must be tough yourself.

—Old Chinese Proverb

Alexander Tereshkin and Rafal Wojtczuk, from the Invisible Things Labs of Poland, introduced the concept of "Ring -3 rootkit" at the 2009 Black Hat conference in Las Vegas.[1] They presented an attack against host memory through a rootkit installed on Intel's management engine. Audiences, many hearing about the management engine for the first time, were impressed by the sophisticated attack. People asked: If the embedded system itself is buggy, how could users trust it to safeguard users' valuable assets?

The security and management engine is a small computer, with its own processor,[i] memory, and nonvolatile storage. It has the capability of performing certain tasks that do not require high bandwidth or data throughput. It acts as a helpful assistant to the main operating system, to carry security sensitive operations that are too risky to be executed on the more exposed main processing environment. In addition to security, the engine also enables platform manageability features and capabilities, such as AMT (Active Management Technology; see Chapter 2).

Due to the nature of the engine, in order to perform its assigned tasks, the engine has to communicate with the host operating system and the CPU, and access the host memory. For certain cases, the engine has even more privileges than ring 0 software.

As such, the engine itself becomes a possible security backdoor and an interesting target of hackers. Sophisticated attacks may be able to exploit the engine's vulnerabilities, if they exist, and leverage its wide range of privileges to attack against the host system.

[i]In this chapter, *processor* refers to the engine's processing unit. The system's main processor is referred to as a *CPU* (central processing unit).

Therefore, making it strong and robust against attacks is *the* fundamental goal when building the engine. But how is the goal achieved? This chapter reveals the techniques deployed to safeguard the security and management engine from attacks. Note that descriptions of techniques in this chapter are based on the latest engine release for 2014. Security is a progressive effort for the engine. Some of the latest safeguarding features may not be available on older versions of the engine.

The security and management engine is equipped with powerful privileges, which are necessary for the engine to perform defined security functionalities. The embedded engine is not restricted by security measures enforced by the user's operating system, Windows, Linux, or Android. The engine is able to access virtually the entire host memory space with the exception of certain system-reserved regions. The engine can also communicate with the CPU of the platform and instruct the CPU to perform specific operations. For power management, the engine has the capability to instantly power down the entire platform.

However, the security and management engine is not a black box to the host. The engine reports its status at runtime to the host via a register that is read only by ring 0 drivers of the host operating system.

Access to Host Memory

Recall that the HECI (host-embedded communication interface) introduced in Chapter 3 is a communication channel between the engine and the host. However, it suffers from narrow bandwidth—only a small amount of data can be transmitted per transaction. Due to such restrictions, HECI is commonly used for delivering control and management commands, but not bulk data.

Many applications on the engine have the need to exchange large amounts of data between the engine and its software counterparts running on the host operating system. For example, for content protection usage, the engine must first copy encrypted video and audio frames from the host to the embedded memory, and then perform decryption. A movie can have hundreds of thousands of frames, and they must be processed at high speed to ensure smoothness of the playback. Another example: the wireless LAN (WLAN) embedded application must copy network traffic data to the host memory and send it through the WLAN adapter.

To support such uses, the backbone of the engine contains dedicated DMA (direct memory access) hardware that copies data between the host memory and the embedded memory. The engine's firmware kernel is the only entity that manages DMA operations between the host and the engine through the DMA devices. Embedded applications call a kernel API (Application Programming Interface) to request DMA to and from the host memory. Host memory is referenced by its physical address.

Obviously, reading and writing arbitrary host memory is a superior privilege that, if abused, can result in serious security consequences. The attack against the engine presented by Alexander Tereshkin and Rafal Wojtczuk exploited a buffer overflow bug in the BIOS[2] and a critical design flaw in the engine, and managed to turn the engine into a rootkit that can write to arbitrary host memory.

To respond to the attack, in addition to fixing the BIOS' buffer overflow bug and correcting the engine's design flaw, several hardening measures have also been implemented on the engine.

- *Small DMA driver*: Have a small "privileged" component, named "DMA driver," in the firmware kernel manage the DMA devices. The kernel is logically isolated from other firmware modules. The kernel is subject to more stringent code review and validation to ensure it is free of bugs.
- *Restrictive access control*: The DMA access is not granted to all firmware applications. An application must show justified reasons to invoke the DMA engine. The list of applications that are allowed DMA access is predefined and hardcoded in the DMA driver. At runtime, the DMA driver identifies the caller and makes sure it is on the white list, before fulfilling the request.
- *Restrictive memory range control*: For a firmware application that is allowed DMA access, the logic for determining host memory ranges to be accessed must be a separate component that is logically isolated from the rest of the application. Just like the DMA driver, such components are subject to more stringent code review and testing to ensure they are free of bugs.
- *Integrity protection on "borrowed" memory*: The firmware reserves a portion of DRAM (dynamic random-access memory) and uses it as secondary memory at runtime. The "*borrowed*" memory is protected for integrity and confidentiality against attacks from the host.
- *Blocked access to certain system memory*: The engine's DMA devices are not allowed to read or write certain system memory; for example, the memory regions reserved for VT-d[2] (Virtualization Technology for Directed I/O) and SMM[3] (System Management Mode).

Communication with the CPU

Some firmware applications running on the security and management engine coordinate with the CPU to perform certain functionalities that involve both the engine and the CPU.

On SoC (Systems-on-Chip) systems, the data between the embedded engine and the CPU is transmitted over the Intel on-chip system fabric (IOSF). The engine's firmware was designed based on the presumption that IOSF is insecure; that is, third parties may eavesdrop the data travelled on IOSF. Therefore, no secrets or keys may be sent in the clear between the engine and the CPU. Secrets are always encrypted before transmission.

On big-core systems, the data between the engine and the CPU is transmitted over the DMI (Direct Media Interface) link. Similar to the case of IOSF, the DMI link is not trusted.

Like the DMA driver, there is a privileged "IOSF driver" and "DMI driver" in the engine's kernel that centrally manages access to the CPU. Applications that are allowed to access to the CPU are predefined, and such privilege is granted on a "need to have" basis.

Triggering Power Flow

The engine's power management unit is able to trigger power state transitions for the engine and the host. Some applications running on the engine perform power transitions at defined scenarios. For example, anti-theft[ii] must unconditionally shut down the platform without notifying the host or asking for the user's consent when it finds the system in a stolen state.

Another usage model of power transition is when an attack is detected. The engine may instantly shut down the platform to terminate the attack and prevent secrets leakage.

Security Requirements

Setting requirements is the first step for the product architecture and design. For an embedded system such as the security and management engine, security requirements are as important as, or even more important than, functional requirements.

At a high level, the engine is made up of a kernel and multiple applications running on top of the kernel. This section discusses general security requirements that must be followed by the kernel and all applications. In addition to these requirements, individual modules should define their own security requirements. For example, a basic requirement for the content protection application is never to expose its device private key or clear premium content to the host.

General security requirements used by the NIST's Common Vulnerability Scoring System[5] (CVSS) include:

- Confidentiality
- Integrity
- Availability

In addition, there is a basic guideline for realizing security: *Never rely on security through obscurity*.

When designing security hardening features for the engine, it is always assumed that all firmware source code and internal architecture documentation may be obtained by attackers. The engine's security design principle is to harden the product by applying proven cryptography and security primitives, rather than rely on hiding secrets in the code or documents.

Confidentiality

The security and management engine treats code segments and noncode segments differently when applying confidentiality protections. The code segment, also known as a text segment, is read-only and contains executable instructions. Noncode segments include data, heap, bss, stack, and so on. In this chapter, noncode segments are referred to as *data segments* for the sake of simplicity.

[ii]Anti-theft is an Intel technology for protecting data on mobile devices from being stolen. Intel has announced the termination of the service by the end of January 2015.

The engine processes many different secrets of high value in its data segment. Examples include:

- EPID (enhanced private identification) private key (see Chapter 5 for details)
- TPM (trusted platform module) endorsement key (see Chapter 7 for details)

Secret data must be kept private, at runtime and at rest. The engine has dedicated internal memory (static random-access memory or SRAM) as level-2 cache for storing runtime data and processor instructions. The memory is not accessible from the outside world.

As the internal SRAM is expensive and limited, the engine also "borrows" the system DRAM as level-3 cache and uses it to temporarily store memory pages that are not recently accessed by the processor. The DRAM is considered insecure. All data pages swapped to the DRAM, whether they contain secrets or not, are encrypted with a 128-bit AES-CBC key.

To provide confidentiality protection for secrets at rest, during manufacturing, each instance of the embedded engine is installed with unique security fuses. The kernel derives a 128-bit AES key at every boot. The key is used to encrypt nonvolatile data before the data is stored on the SPI (Serial Peripheral Interface) flash.

For applications that interact with the outside world (software programs running on the host, CPU, network, and so on), the communication channels are treated as open channels that malware can read and alter. Therefore, secrets must be protected by appropriate encryption algorithms or protocols, such as TLS[6] (Transport Layer Security). Individual applications are responsible for the protection.

What about the code segment? Due to major performance costs of encrypting code, the security and management engine does not protect confidentiality of its compiled binary image. By design, the firmware binary should not contain secrets, and hence it is not encrypted or obfuscated in any form. Note that lossless compression may be applied to the code.

The firmware binary, in its compression form, is stored on SPI flash in cleartext. At runtime, the code segment is not encrypted when it is paged out to DRAM.

Admittedly, advanced hackers have successfully reverse-engineered and disassembled the engine's firmware binary. However, knowledge of source code is not deemed a harmful threat, because no secrets or keys are ever hardcoded in the code, and the architecture and robustness of the engine does not rely on security through obscurity.

Integrity

The integrity protection makes sure that the target being protected has not been altered unexpectedly due to corruptions or attacks. Several algorithms are common choices for integrity assurance.

- *Digital signature, such as RSA and ECDSA*: The owner of the raw data signs the data with her private key. The signature is then appended to the raw data. Any entity that knows the corresponding public key can verify the owner's signature on the data. Because operations of digital signature are relatively slow, it is usually used for signing small amounts of data.
- *Keyed hash*: The owner of the raw data calculates a digest with a secret key. The digest is then appended to the data. Any entity that knows the secret key can verify the digest of the data.
- *Plain hash*: The owner of the raw data calculates a digest without a key. The digest is then appended to the data. Any entity can verify the digest of the data.
- *CRC (cyclic redundancy check)*: CRC is not a cryptography algorithm but an error-detecting scheme, which is intended to detect accidental changes to data, rather than intentional attacks. A short (for example, 32 bits) parity check value is calculated using the CRC algorithm and attached to the raw data. On retrieval, the same calculation is repeated and the result is compared with the appended parity.

The kernel of the security and management engine provides interfaces for all aforementioned algorithms for applications, to protect their data's integrity.

For an embedded system, integrity of the code segment is also a critical consideration. It is a requirement that the security and management engine's processor and hardware executes only unmodified instructions that were signed by Intel or a designated entity. The design flaw exploited by Alexander Tereshkin and Rafal Wojtczuk was lacking integrity protection for the code segment, allowing injection and execution of malicious code that is not endorsed by Intel.

More details about the approach for protecting the integrity of the engine's code segment are discussed later in this chapter.

Availability

Availability refers to the accessibility of the services provided by the embedded engine and the platform. Note that the availability requirement of the engine applies to the entire system, including the host. In other words, the engine must not cause the host to crash or become unavailable.

The exact requirement of availability varies depending on the attacker's privilege.

- If the attacker has physical access to the platform, then availability is not a consideration. With physical access, one can destroy the system with a hammer.

■ **Note** The anti-theft application is an exception—it must be available to function even if the attacker has physical access to the platform.

- If the attacker has local access—that is, he can install malware on the host operating system—then he shall not be able to disable, reset, or turn off the embedded engine.
- If the attacker has network access, then similarly to local access, he shall not be able to disable, reset, or turn off the embedded engine.

The general guideline regarding availability is that malware or virus on the host system or network shall not be able to mount denial of service (DoS) attacks against the engine. This requirement implies that the engine's external (such as HECI and network) interfaces must be robust. They must reject malformed input gracefully and handle large amount of requests properly. Under any circumstances, an external input should not cause the engine to crash. Note that the engine supports multiple usages and features that are running over the kernel. Security protections of one feature must be protected from compromise by users of another service. For example, an AMT administrator shall not be able to influence EPID operations.

The anti-theft application has its unique functionality, and hence, special requirement about availability. The definition of availability for anti-theft is opposite to what availability normally means. By design, it must enforce unconditional shutdown of the platform when the system is detected to be in the stolen state.

In the stolen state, the thief (attacker) possesses the platform and has physical access. In this case, anti-theft must continue to be available and function normally by enforcing the platform shutdown per defined policies. The attacker may physically destroy the platform and render it unusable, which does not violate the availability requirement of anti-theft.

Another important requirement is the availability of the host. Because the embedded engine is able to trigger instant shutdown of the system, malware may exploit firmware vulnerability to shut down the computer locally or remotely, realizing an annoying DoS attack. This is an ungraceful shutdown, and all unsaved user data will be lost. The attack may launch repeatedly right after reboot and essentially turn the computer into a brick.

The Sasser worm of 2004 is a notable example of how costly DoS attacks can be. The author of the worm reverse-engineered a patch released by Microsoft that fixed a buffer overflow bug in Windows 2000 and XP, and discovered the bug. The worm exploited the vulnerability on computers that had not installed the patch. The worm allowed remote execution of code on the host without the knowledge of the user. In the United States alone, the shutdown of computers due to the Sasser worm resulted in a damage of approximately 15 billion US dollars.[7]

Threat Analysis and Mitigation

The threat analysis involves applying the general security requirements—confidentiality, integrity, and availability—to the architecture and design of the security and management engine.

This section reviews most critical threats that are considered during the development of the engine, and the corresponding security measures and mitigation plans implemented by the engine.

Load Integrity

There are two physical locations at which the firmware image of the security and management engine are stored:

- The boot loader is stored in a ROM (read-only memory). Thanks to the nature of ROM, this small portion of code is considered intact. Mitigation against altering or injecting to the code in ROM is out of scope. The ROM is the root of trust of the embedded engine. Note that physical tampering and fault injection attacks are out of scope.
- The rest of the firmware image is stored in SPI flash together with BIOS and other firmware ingredients of the system. Different products support different sets of features and applications. Depending on the product, the size of the engine's firmware ranges from 1.5MB to 5MB.

Although the flash part is supposed to be locked down at manufacturing, in security modeling, it is assumed that the chip can be replaced and the lockdown mechanism can be circumvented by attackers. Therefore, when the boot loader in ROM is loading the image from the flash, it must be confident that the loaded code has not been modified.

The firmware image on flash is signed by Intel. The signing algorithm is 2048-bit RSA with an SHA-256 and a PKCS#1 padding scheme. The signature is not on the entire binary image of a few megabytes, but on a small manifest for the binary.

The manifest contains information for all firmware modules. A module can be the kernel or an application such as anti-theft, content protection, and so on. Among all the information of a module described in the manifest, the most critical, security-wise, is the SHA-256 digest of the module. The SHA-256 digests of all modules are digitally signed.

Here is the flow of building a firmware image:

1. Compile all modules.
2. Calculate SHA-256 digests for all compiled modules, respectively.
3. Fill in the manifest header. The header includes fields such as:
    a. Firmware version number
    b. Firmware security version number

c. Size of the header

d. Number of modules

4. Apply compression algorithms to modules, if applicable. There are three options to choose from for a given module:

   a. No compression

   b. Huffman compression[8]

   c. LZMA[9] (Lempel-Ziv–Markov chain) compression

   Decompression is performed by the boot loader in ROM during loading. The engine has dedicated hardware logic to support Huffman, so the Huffman decompression is relatively fast. For an LZMA-compressed module, the decompression is carried out by firmware logic located in ROM. As it is a firmware implementation, the decompression is slower than that of the Huffman decompression. However, the adaptive LZMA enjoys a higher compression ratio than Huffman, which uses a hardcoded static dictionary. There is a tradeoff between binary image size and decompression performance at load time. In general, kernel components that impact load time choose no compression or Huffman compression for performance reasons, and applications normally use LZMA. Note that the data after decompression is still not trusted, so an attack on corrupting the decompression results is equivalent to flash corruption.

5. Fill in all module entries in the manifest. A module entry has information such as:

   a. Name

   b. SHA-256 digest

   c. Location of the compressed binary in the image

   d. Compression algorithm

   e. Compressed size

   f. Uncompressed size

   g. Entry point address

6. Fill in the RSA public key (values of the 2048-bit n and the 32-bit e) that will be used by ROM to verify the signature during loading.

7. Sign the manifest using the RSA private key and place the signature in the manifest. The 2048-bit signature is generated on the entire manifest data exception for the RSA public key and the signature itself.

8. Append all modules after the manifest at their proper locations specified in the module entries.

The firmware security version number in the manifest header is an important field for managing firmware update or downgrade for cases where vulnerability is found and patched. Figure 4-1 illustrates the structure of the manifest.

Manifest header	Header type
	Header length
	Firmware version number
	Firmware security number
	Header size
	Number of modules (m in this example)
	Etc.
Crypto block (RSA signature does not cover this part)	RSA public key n
	RSA public key e
	RSA signature
Module #1	Module name
	SHA-256 hash value
	Address
	Compression type
	Compressed size
	Uncompressed size
	Entry point address
	Etc.
Module #2	...
...	...
Module #m	Module name
	SHA-256 hash value
	Address
	Compression type
	Compressed size
	Uncompressed size
	Entry point address
	Etc.

Figure 4-1. *Manifest floor plan*

During boot, the embedded engine's ROM initializes internal memory and copies the firmware image from the flash. The first thing it loads from the flash is the manifest.

Here is the boot loader flow in ROM:

1. Read the RSA public key from the manifest.
2. Calculate the SHA-256 hash on the RSA public key and compare the resulting digest with the hardcoded digest in ROM. If they do not match, then the image is corrupted and will not be loaded.

   When ROM is created, the SHA-256 digest of the RSA public key is hardcoded in the code. The reason for hardcoding the 256-bit hash of the RSA public key, and not the complete 2080-bit RSA public key itself, is to save space in ROM.
3. Verify the digital signature of the manifest using the public key. If the signature verification fails, then the image is corrupted and will not be loaded.
4. Check validity of the manifest header, such as the firmware version.
5. Load the first firmware module by copying its binary from the flash. The first module is usually named "Bringup" or "Kernel". If the module is compressed, then perform decompression.
6. Calculate the SHA-256 digest on the decompressed module and compare with the corresponding hash value in the manifest. Note that at this point, the hash value in the manifest has already been verified by the RSA signature at step 3. If the digests do not match, then the image is corrupted and will not be loaded.
7. Once the first module is loaded, ROM hands the control to the "load manager" component of the first module, which will continue to load other modules listed in the manifest.
8. To load a module, the load manager copies the module's binary from the flash and performs decompression, if required. Then the load manager calculates the SHA-256 digest of the module and compares it with the digest in the manifest. If they do not match, then there are two options:

   - Stop loading this module and continue to load the next module, or
   - Unload all modules that have been loaded and halt the engine's processor

   The option taken depends on whether the module is fault-tolerant or non-fault-tolerant. Failure to load a fault-tolerant module does not break the engine's functionality or impact other modules of the engine. On the other hand, all non-fault-tolerant modules are required for the engine to function properly.

The ROM flow for loading the engine's firmware is depicted in Figure 4-2.

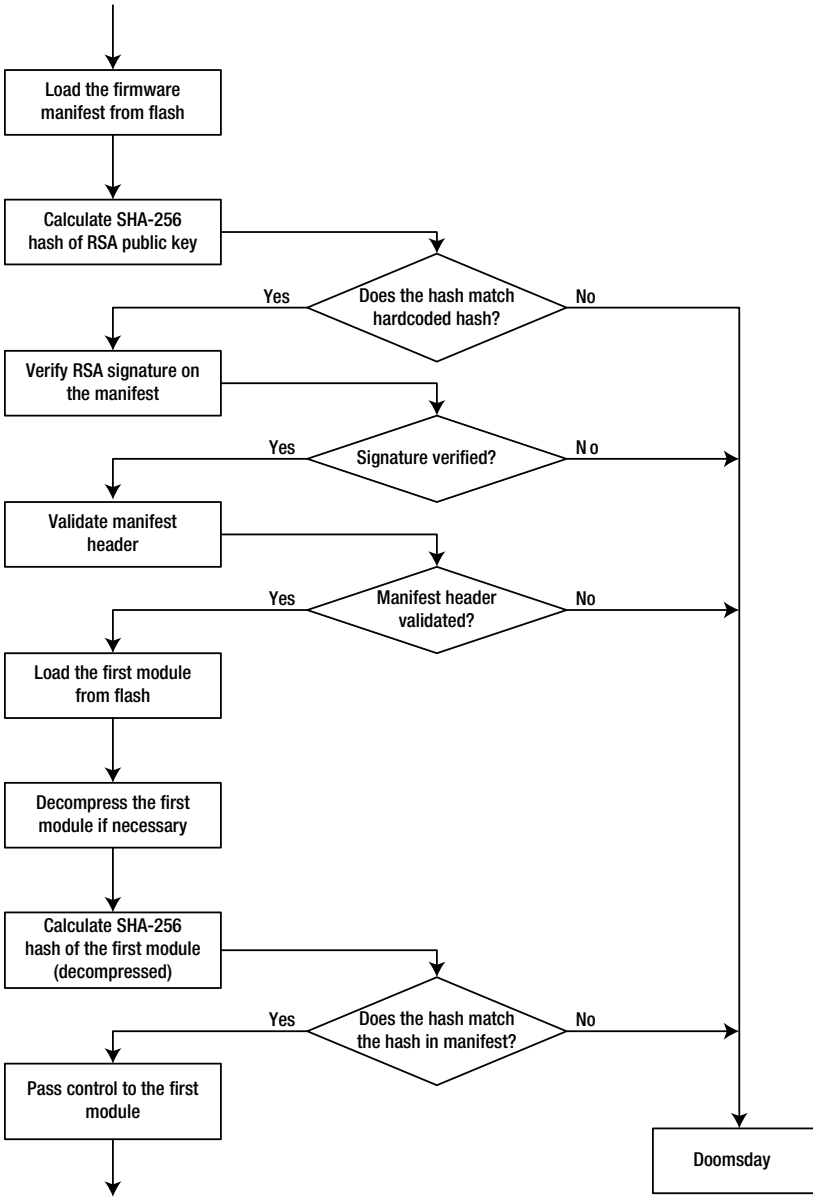

Figure 4-2. *ROM flow for loading firmware*

Note that for a compressed module, its hash in the manifest is calculated on its decompressed binary instead of the compressed binary. This means that the boot loader first decompresses the module, places the decompressed module in the engine's internal memory, and then verifies its integrity.

Does something seem suspicious? Yes. Unverified compressed binary is being placed in memory, at least temporarily. The binary is then decompressed to the internal memory. If the compressed fault-tolerant module is altered by an attacker, then it could overflow the buffer allocated for the decompressed module and overwrite other regions of the internal memory, making a code-inject attack possible. So hashing a decompressed module is arguably a poor security design practice and prone to vulnerabilities. To address the issue, the implementation must make sure that the buffer allocated for decompressed data is not overrun by the decompression algorithm.

A better design from the security perspective would be to hash the compressed form of the module. However, there is a major drawback of this option: memory consumption. The entire compressed module must be copied into internal memory before the decompression begins, and memory must be reserved for both the compressed module and the decompressed module.

On the other hand, if the hash is for the decompressed module, then there is no need to copy the compressed module into memory. The boot loader simply reads from the flash the compressed module in fixed-size chunks, and then performs decompression for the chunks as they come in. The decompressed module in the internal memory is then verified against the hash value specified in the manifest.

When architecting a computer system, there are two conflicting factors to consider, one being performance and resource consumption, and the other being security. There is almost always a tradeoff between the two sides. For systems where resources are not a major concern, it is usually better to be safe than sorry and give more weight on security. For embedded systems, however, due to the limited computing resources available, the decision is sometimes more difficult to make. It requires designers to dive deep into the threat analysis and risk assessment.

Memory Integrity

For the security and management engine, the level-1 cache is inside the processor. The engine has dedicated internal memory that serves as the level-2 cache. The capacity of the level-2 cache varies, depending on the product, and ranges from 256KB to 1MB. In the security modeling, the level-1 and level-2 cache memory is considered immune from external attacks. No encryption or integrity protection is applied.

But the embedded engine requires more runtime memory to run its applications. A small region of the system's DRAM is "borrowed" by the engine and used for the purpose of paging. The size of the borrowed memory ranges from 4MB to 32MB, depending on the product.

The embedded engine uses the borrowed DRAM for temporary volatile storage only. The engine's processor cannot directly reference addresses in DRAM, execute code from DRAM, or modify data in DRAM. When a page in DRAM needs to be accessed by the processor, the engine's paging unit has to first bring it into the internal memory.

During boot, the BIOS reserves a small portion of DRAM and notifies the security and management engine of its address and size. The BIOS hides this portion of DRAM from the operating system running on the host. From then on, the engine has exclusive control and access to this region. The host is not supposed to address, reference, or access the region.

However, hackers have shown that breaking into the reserved DRAM region is not impossible. The attack presented by Alexander Tereshkin and Rafal Wojtczuk successfully injects code into the reserved region. The injected code is later paged in by the engine and executed. This attack was possible because on Bearlake MCH (Memory Controller Hub), the management engine lacks integrity protection for the reserved region of DRAM.

How is the problem tackled in later generations of the security and management engine? Checksum is introduced for paging:

1. Before moving a page from the internal memory to the reserved DRAM region, calculate a checksum of the page and store the checksum in the internal memory.
2. The content of the page is not supposed to change while it is out in the DRAM.
3. After moving a page from the reserved DRAM region to the internal memory, calculate the checksum of the page again and compare with the stored value calculated before. If the two values do not match, then the page has been altered. Although this is possibly due to a memory corruption, for defensive security design, the security and management engine treats it as an attack and triggers an instant shutdown of the platform, which includes the engine itself and the host.

When looking for the right checksum algorithm, several conditions were considered:

- The algorithm must be extremely simple and fast. Since paging is a very frequent runtime operation, the speed of paging plays a significant role in the engine's performance. Latency of paging must be minimized, as it negatively impacts the user's experience.
- The checksum must be small in size, because the internal memory space is limited and expensive. The more internal space is assigned to checksum storage, the less space is available for running programs.
- The algorithm must be able to detect alteration of pages in DRAM with a high level of confidence.

Digital signature is ruled out immediately, as it is too slow to meet the performance and storage requirements outlined.

Next candidates are hash and HMAC. Velocity-wise, they are much faster to calculate than digital signature schemes. Also, the security and management has a hardware cryptography engine for expediting hash and HMAC. Security-wise, they are NIST-approved algorithms that offer proven strength of integrity assurance. But they are still not optimal because of two reasons:

- The size of the digest is too large to fit in the internal memory. If the reserved region is 16MB and page size is 4KB, then there are 4096 entries. Using SHA-1, the size of internal memory required for storing all digests is as much as 80KB. Additionally, as will be discussed later in this section, there is other metadata that must be stored in the internal memory for a page entry.
- The speed of calculation is not fast enough to support runtime applications that require high throughput, such as AMT.

Now the only candidate is the CRC algorithm. It is simple and fast to calculate. The checksum is only 32 bits long. All that makes it a good choice from the performance perspective. What about security?

CRC is an error-detecting code. It does not use a key and it is not cryptographically strong. Imagine a naïve attack scenario: the hacker reads a page from the reserved DRAM region and calculates its CRC checksum. He then modifies the page content such that the checksum remains unchanged. For a 4KB page and 32-bit checksum, finding different pages with the same checksum is rather trivial.

So, it seems none of the standard integrity protection algorithms has characteristics to satisfy all requirements of the security and management engine. To address the problem, Intel's cryptographers have designed a proprietary algorithm specifically for paging integrity. The algorithm is based on binary polynomial operations. The input includes:

- 4KB or 1KB of raw page data
- 256-byte secret key

The output is a 32-bit integrity check value (ICV), which must be kept secret.

During the first time the security and management engine boots and before paging is enabled, the engine generates a 256-byte random number and writes it to the registers of the ICV generation hardware logic. The engine also stores the random number on flash as a secret blob. This number is used as the secret key input to the ICV algorithm.

During the following boots, the key is retrieved from the flash and reused. Although regenerating a new key randomly at every boot is apparently more secure, it is experimentally shown that generating 256 bytes of random data from the engine's hardware RNG is slower than reading a blob from the flash. For most computing systems, the boot time is a critical performance benchmark.

However, there is one case that the ICV key will be regenerated. Before moving a page out of the internal memory, the paging engine in the kernel calculates the page's ICV value and saves the resultant ICV in a preallocated region of the internal memory. The ICV calculation is performed by dedicated hardware logic. Later, when bringing a page into the internal memory, the same calculation is repeated and the result compared with the saved value.

What if the comparison fails? When the security and management engine feels that "something is wrong," several different actions can be considered as response. The firmware designers must decide what actions to take when something is wrong. The questions to ask are as follows:

- Is the error more likely a result of a firmware or hardware bug, or is the error more likely due to an active attack?
- Is it possible to recover from the error without leakage of secrets and assets?

Because of its criticality, all firmware and hardware components involved in the paging operation are reviewed and validated thoroughly. Furthermore, DRAM failure is very rare thanks to improved error-correcting and other technologies deployed in modern DRAM devices. Given these facts, when an ICV check failure occurs, the engine has very high confidence that it is due to an attack that is attempting to change the page being brought in from the DRAM. The most effective response to terminate the attack and prevent loss of assets is to shut down the platform immediately and ungracefully.

Before shutting down the platform, the engine deletes the blob that stores the ICV key from the flash. At the next boot, the engine will generate and use a new key.

Admittedly, the algorithm is not as strong as a standard hash, but it is good enough to protect the engine. With this proprietary algorithm, page alternation or replacement attacks become very difficult to mount.

As the ICV is a 32-bit secret, and the key is also secret, an attempt at random page alternation has a success probability of only 1 in 2^{32}. A random attempt will fail almost definitely, and as a result, the platform is rebooted and a new ICV key is utilized. This means that the attacker cannot learn from failures, and his prior failed attempts do not increase the chance of future success. All attempts have a success probability of 1 in 2^{32}, no matter if it is the first or the one thousandth attempt.

Another important design to make the attack even harder is that the engine keeps the ICV secretly. Furthermore, a platform reboot following a failed attempt takes at least a few seconds to complete, which substantially slows down automation. As the ICV of a page is unknown, hackers cannot simply perform the page alternation attempts "offline" without actually running the engine.

As a result, altering a page and not being detected by the embedded engine is practically impossible.

Checksums must be kept secret. A straightforward design is to keep the checksums for all pages in the internal memory. This method consumes valuable memory space. To save memory space, the security and management engine also swaps pages that store checksums to reserved DRAM region. The checksums for such pages are always stored in the internal memory.

When a page fault happens, the paging engine looks for the checksum of the page in the internal memory. If the checksum is not found, that means the checksum is out in the DRAM also. In this case, the paging engine brings the checksum page into memory first, and then brings the actual page of the page fault into memory.

For this design, handling a page fault may require two pages being swapped into memory, which seemingly will degrade performance. But the fact is, the opposite occurs. Experiments show that with comprehensive victim (a page that is selected to be swapped out to DRAM) selection heuristics, this design actually improves performance because there are fewer checksums occupying memory, and hence more memory is available as cache.

Memory Encryption

Besides integrity, confidentiality is also a requirement for data pages while they reside in the reserved DRAM region. A page is encrypted before being moved out to the DRAM. The ICV is calculated on the encrypted page. Pages that contain only code segments require protection for integrity but not confidentiality.

The algorithm used for encrypting data pages is 128-bit AES with CBC mode. During boot, the AES key for encrypting pages is derived from security fuses. The key is unique per part, as the fuses are unique. The key is stored in the internal memory and never paged out to DRAM.

Since the IV (initialization vector) for CBC mode must be unpredictable, the IV for encrypting a page is randomly generated every time the page is about to be moved to DRAM. The IV is stored together with the ICV.

Task Isolation

An *embedded system* is a computer system designed to realize dedicated and specific functions with computing constraints. The system includes hardware and firmware that runs on the hardware.

Embedded systems usually suffer from resource constraints (limited computing horsepower, memory, storage space, and so forth). An embedded system with a single-threaded or a multithreaded real-time operating system (RTOS) can run multiple processes. On the security and management engine, process is also referred to as *task*.

In an embedded system that runs multiple processes (tasks) without isolation, successful attack or compromise against one or more applications may result in the attacker gaining execution privilege and secrets of the peer applications. This is a critical security problem for embedded systems.

Process isolation as a security measure is widely supported by modern operating systems such as Windows, Linux, and Android. Is the same concept applicable to embedded systems? Intel's security and management engine resolves the problem by applying innovative task isolation techniques. The task isolation is the most involved and comprehensive security measure on the engine. This section covers the details of the technique.

Deploying task isolation on the engine has been an evolving effort. There was no task isolation for the first generation of the engine, as the size of the firmware was relatively small at that time, and all kernel and applications were developed in-house by Intel. As the number of applications running on the engine increased, isolation became a must-have security measure.

As the first step, the engine's firmware was split into two tasks—privileged and nonprivileged:

- The privileged task, also known as the *kernel*, consists of modules that manage critical system resources and handle secrets. They include the boot loader, kernel, hardware drivers, power flow management, EPID manager (see Chapter 5 for details), and so on.
- The nonprivileged task consists of the remainder of the firmware modules; for example, applications like AMT and anti-theft.

The logical separation between privileged and nonprivileged tasks is enforced by the privileged task and hardware. The hardware backbone of the engine supports two modes of operation: privileged mode and nonprivileged mode. Different access rights to hardware devices and other system resources are granted based on the mode in which the firmware is actively running.

In newer versions of the security and management engine, the number of embedded applications keeps growing. The number exceeds ten on the engine shipped with big core processor in 2013 (codename Haswell). With this many applications, the size of the engine's nonprivileged modules becomes considerably large. Consequently, risk of security bugs and vulnerabilities rises.

How to realize task isolation for multiple tasks in a hardware environment that supports only privileged and nonprivileged tasks? The trick is to treat and protect all nonprivileged tasks that are not actively running as privileged tasks, so that the running nonprivileged task cannot compromise them.

Asset Protection

The task isolation technique implemented by the engine makes sure that bugs in one task are restricted to its own task and do not affect any other tasks. In other words, even if the bug is exploited by attackers, other tasks are immune and safe.

The assets of a task to be protected from other tasks include but are not limited to the following:

- Memory
- Nonvolatile storage
- Hardware devices
- Synchronization objects: thread, semaphore, mutex, queues, and so forth

An asset belongs to one and only one task during its lifetime. The owner is normally the creator of the asset. The ownership cannot be transferred to another task.

The central governing component, kernel, manages all system resources. It is responsible for implementing and enforcing task isolation for nonprivileged modules. The kernel is a hybrid component of firmware and hardware. The interface of the kernel is minimized to reduce the attack surface.

The kernel provides critical and system level services to nonprivileged components. These services include: cryptography algorithms, memory management, nonvolatile storage, DMA, power management, and so on. For protected assets owned by individual tasks, the kernel exposes API for the tasks to call and manipulate.

For example, nonvolatile secrets stored on flash are assets of their owning tasks. The kernel has APIs for creating, writing, reading, and deleting the data. Another example: semaphore is an asset of its owning task. The kernel has APIs to create, get, put, and delete a semaphore.

Figure 4-3 demonstrates the kernel's flow of handling a call from a nonprivileged task for asset manipulation. A few important facts to note:

- The kernel is threadless and all kernel API functions run in the caller's thread.
- A thread is always associated with one and only one task.
- Metadata of threads and other assets for all tasks is stored in the privileged memory and cannot be modified by nonprivileged tasks. The metadata of an asset includes the ID of the owning task of this asset.

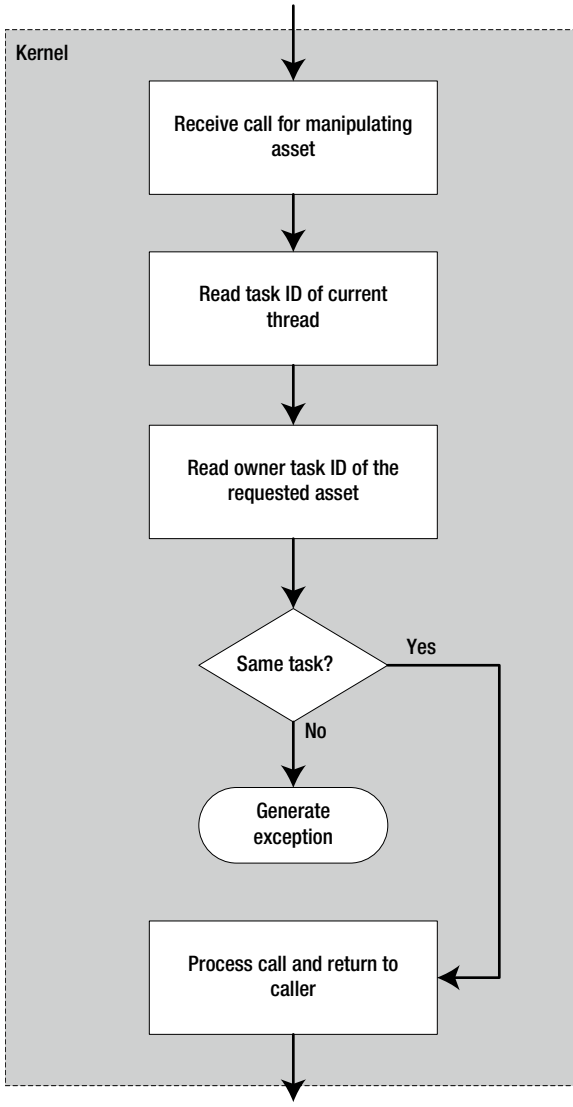

Figure 4-3. *Asset (nonvolatile data, synchronization objects, and so on) manipulation control flow*

As you can see in Figure 4-3, the kernel makes sure that the asset being accessed belongs to the same task as the caller's thread—that is, an application is not allowed to access another task's assets through kernel APIs. Such a request is considered an attack and will trigger exception. If a task has legitimate reasons to access assets of another task, then it must do so through the inter-task call mechanism.

Memory Manager

The memory manager, a component in the kernel, is responsible for the following:

- Managing the embedded system's memory space
- Creating a dedicated memory pool for each task (a task can only access its own memory region)
- For `malloc()` calls, allocating memory only from the calling task's memory region

The embedded engine's memory is divided into multiple regions as overlays. The kernel has read/write access to all memory regions. There is no memory region that can be accessed by more than one nonprivileged task. The size of a memory region is determined by the actual usage model of the owning task. A task can be assigned multiple memory regions with different properties—for example, one region that can be accessed by both the processor of the engine and the DMA devices, and another region that is only accessible by DMA.

Figure 4-4 shows a conceptual example of three tasks in 1MB memory space and their overlays.

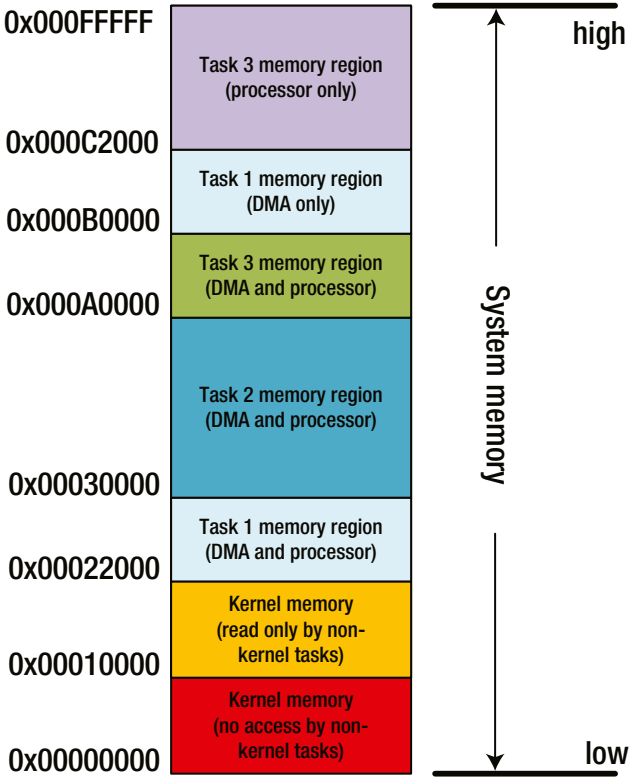

Figure 4-4. *Memory overlay*

Thread Manager

The single-threaded or multithreaded thread manager is also a component in the kernel. It manages threads and schedules threads to run.

One and only one thread is actively running at any moment. A thread is associated with one and only one task throughout the lifetime of the thread.

At runtime, the system determines whether requested assets/resources can be accessed based on the task of the currently running thread. At thread switch,[iii] the RTOS examines the owner tasks of the current thread and the next thread, respectively. If the two threads are owned by different tasks, then the RTOS programs the MPR (memory protection range) control register accordingly to predefined values to reflect the restriction applied to the next thread. Figure 4-5 illustrates the flow.

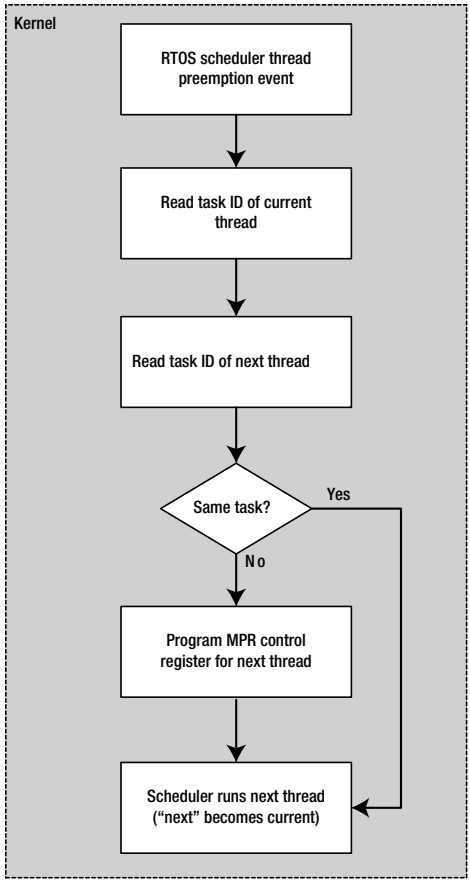

Figure 4-5. *Thread switch flow*

[iii]The scheduler decides to preempt the currently running thread with another thread.

Memory Protection Control

The security and management engine's hardware backbone supports a set of MPRs. The number of MPRs implemented in a specific product depends on the number of nonprivileged tasks. Each MPR consists of a set of three registers:

- Start address
- End address
- Access restriction (assumes one of the following values)
    - No read/write by processor or DMA
    - Read only by processor and DMA
    - No read/write by processor but can be read/write by DMA
    - Can be read/write by processor but no read/write by DMA
    - Other access restrictions as needed

The MPRs enforce the access restrictions applied to the currently running nonprivileged task for the entire memory space. When the kernel is running, MPRs are not enforced because the kernel can access the entire firmware memory space.

On the security and management engine, an MPR control register is introduced for rapidly enabling and disabling an arbitrary set of MPRs. For example, if there are 64 MPRs, then a 64-bit MPR control register is used, one bit for each MPR. If a bit of the MPR control register is 0, then the corresponding MPR is disabled and not enforced for memory access; if a bit is 1, then the corresponding MPR is enabled and enforced for memory access.

During boot, the kernel programs MPR registers for all possible combinations that may be encountered at runtime. The MPR control register will be programmed by the RTOS at runtime upon task switch to realize fast switch between MPR policies of two tasks. This trick eliminates the need for programming the three registers of each MPR for all MPRs at runtime, resulting in significant performance improvements.

For the example, in Table 4-1, nine MPRs are used. The three registers of each MPR are programmed by RTOS, at boot, as follows:

- *MPR#1:* {0x00000000, 0x0000FFFF, no read/write by processor/DMA}
- *MPR#2:* {0x00010000, 0x00021FFF, read only by processor/DMA}
- *MPR#3:* {0x00022000, 0x0002FFFF, no read/write by processor/DMA}
- *MPR#4:* {0x00030000, 0x0009FFFF, no read/write by processor/DMA}
- *MPR#5:* {0x000A0000, 0x000AFFFF, no read/write by processor/DMA}
- *MPR#6:* {0x000B0000, 0x000C1FFF, no read/write by processor/DMA}

- *MPR#7:* {0x000B0000, 0x000C1FFF, no read/write by processor; RW by DMA}
- *MPR#8:* {0x000C2000, 0x000FFFFF, no read/write by processor/ DMA}
- *MPR#9:* {0x000C2000, 0x000FFFFF, read/write by processor; no read/write by DMA}

Table 4-1. *Active Task and MPR Control Setting*

Active task	MPR control register
kernel task	Don't care
Task 1	110110110
Task 2	111011010
Task 3	111101001

At runtime, the MPR register values do not change; the MPR control register is programmed to reflect memory enforcements. The actively running task and the corresponding MPR control register value (the leftmost bit represents MPR#1, and the rightmost bit represents MPR#9) are shown in Table 4-1.

When the privileged kernel is running, MPRs are not enforced.

When task 1 is running, MPR#1: {0x00000000, 0x0000FFFF, no read/write by processor/DMA} is enabled. According to Figure 4-4, memory range from 0x00000000 to 0x0000FFFF belongs to the kernel. Task 1 shall not access this range. This is why the memory access restrictions defined by MPR#1 are enabled and enforced when task 1 is active.

Similarly, MPR#5: {0x000A0000, 0x000AFFFF, no read/write by processor/DMA} is also enabled when task 1 is running. This is because the range of {0x000A0000, 0x000AFFFF} belongs to task 3, and task 1 shall not access it. Likewise, when task 1 is running, MPR#2, MPR#4, MPR#7, and MPR#8 are enforced.

However, memory access restrictions defined by MPR#3: {0x00022000, 0x0002FFFF, no read/write by processor/DMA} are disabled, because this range is owned by task 1, the running task.

Loader

The loader is responsible for loading a task to the memory region allocated by the memory manger.

Figure 4-6 shows the boot flow. The loader in the kernel loads the tasks one after another to their memory regions and initializes the tasks. The main operation that a task performs at start is to create its worker threads by calling the kernel's thread creation function. All threads created are tagged with the owning application's task ID, and it does not change for the lifetime of the thread.

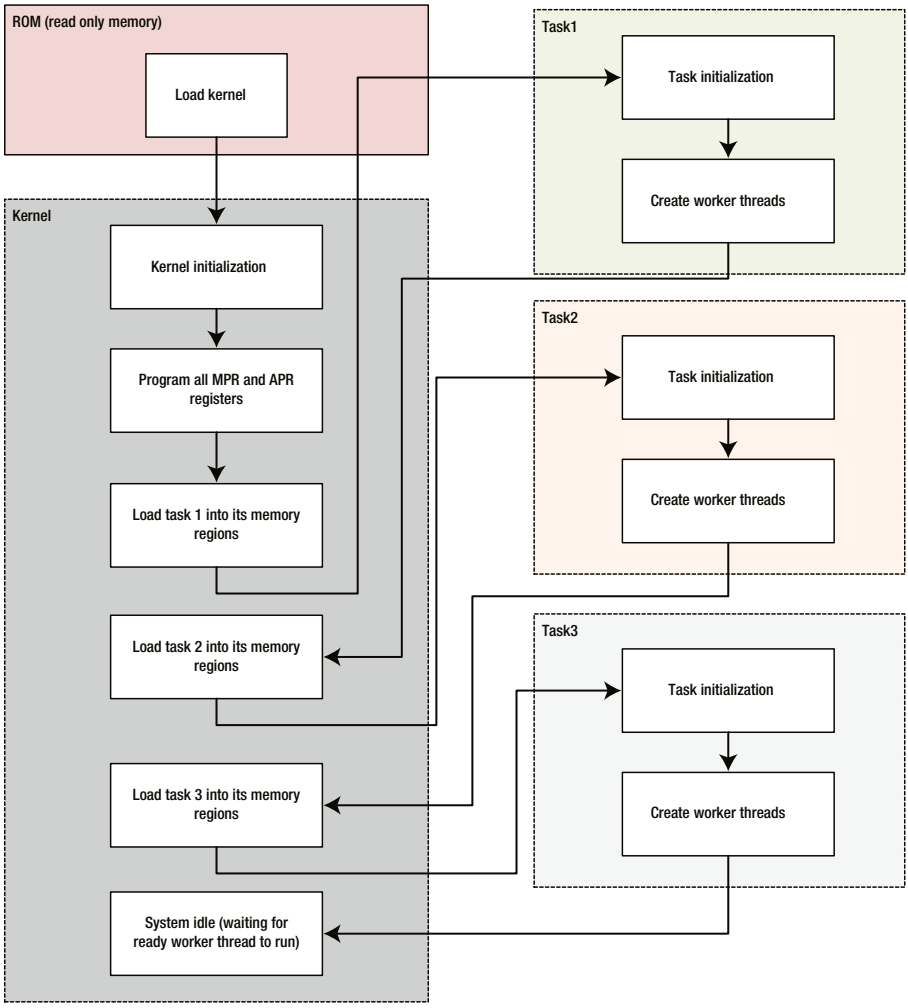

Figure 4-6. *Boot flow*

Inter-Task Call Management

On the security and management engine, a task can provide services to one or more other tasks through an indirect calling mechanism implemented by the kernel. For example, if a task needs to access assets (such as nonvolatile data) of another task, it can do so via the inter-task call mechanism.

Due to memory protection and isolation, direct calling between two tasks is a violation of task isolation and explicitly prohibited. When a task (say, task 1) needs to consume services offered by another task (say, task 2), task 1 invokes kernel's inter-task call API and specifies the function of task 2 to be called. The kernel performs the following steps for an inter-task call.

1. Copy input parameters from task 1's memory to task 2's memory.
2. Call task 2 on behalf of task 1. The kernel will notify task 2 that the caller is task 1. Task 2 can decide whether to serve task 1 or reject the call.
3. Copy the output from task 2's memory back to task 1's memory.
4. Conclude the call.

The inter-task call is a costly operation because the kernel has to copy input and output data between the caller task and the callee task. The design guideline is to minimize the use of inter-task calls and avoid calling other tasks in performance-critical flows.

In Figure 4-7, the dotted line shows that task 1 is calling task 2 through the kernel. Note that all tasks directly consume the kernel and only the kernel. Tasks cannot consume each other directly.

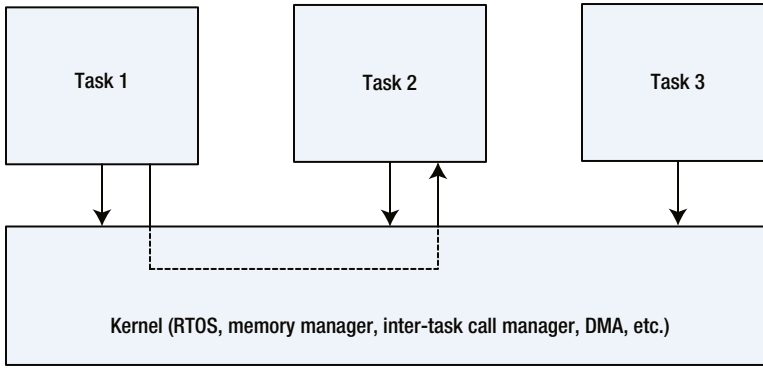

Figure 4-7. *Inter-task call*

Exception Handler

When the kernel firmware or hardware detects access violation, an attack is assumed to be actively undergoing. All threads belonging to the violating task shall be terminated immediately—that is, the task is stopped from running until the next power cycle.

Alternatively, a more aggressive reaction upon access violation is to reset the entire embedded system. This is the approach implemented by the security and management engine.

Nonprivileged Tasks

A nonprivileged task is an embedded application that realizes a specific set of functionalities—for example, playing back a movie.

A nonprivileged task may consume services provided by the kernel and other nonprivileged tasks. A nonprivileged task may also invoke dedicated hardware components. Multiple tasks may exist on an embedded system.

A nonprivileged task is banned from directly accessing other tasks' assets. Such access must be accomplished through the inter-task call mechanism. Access violation results in termination of the violating task or resetting the embedded system.

Firmware Update and Downgrade

The security and management engine supports firmware update and downgrade; that is, replacing the firmware that is currently installed on the platform with another version of the firmware. The firmware update replaces an older version of firmware with a newer version. It is used by Intel to deliver additional features or fix functional or security bugs to the end users. If a newer version of firmware fails to work on a platform, most commonly due to device compliance issues, then the firmware downgrade is used to rollback to an older version of firmware that works on the platform.

The firmware update is launched from a software program on the host. The new firmware can be downloaded from the manufacturer's web site and installed by end users. The new firmware has the same integrity protection mechanism as the current firmware on the platform.

The firmware security number in the manifest header (see Figure 4-1) is used for preventing firmware update or downgrade from a "good" version to a version with known security vulnerabilities. For example, when security vulnerability is found in version A with security number 1, Intel will release version B that fixes the bug. As the new firmware fixes security bugs, the security number will be incremented and B will have a security number of 2.

When a firmware update from A to B is launched, A will check B's security number as it loads the manifest of B. If B's security number is the same or greater than A's, then proceed with the update. If B's security number is smaller than A's, then it is considered a rollback attack (i.e., replacing a patched version with a vulnerable version). In this case, A immediately aborts the firmware update/downgrade flow.

Published Attacks

Ever since its birth in 2006, the management engine has been the target of many hackers and attackers in the computer security community. For white-hat hackers, trying to find and exploit bugs in the engine is an interesting academic research and challenge. For black-hat attackers, successful attacks could generate monetary profit.

To date, the most famous attack against the engine was the one mentioned at the beginning of this chapter: "Introducing Ring -3 Rootkits," published by Alexander Tereshkin and Rafal Wojtczuk of the Invisible Things Labs, at the Black Hat conference in 2009.

"Introducing Ring -3 Rootkits"

There are several components of the attack.

1. Perform literature research and find out the model of the processor used by the engine.
2. Circumvent the flash lock and dump the engine's firmware binary from the flash.
3. Use IDA disassembler[10] to disassemble and reverse-engineer the firmware code.
4. Rollback the BIOS to a version with a known bug that does not lock down memory remapping registers. This vulnerable release of BIOS allows the attack to redirect the engine's reserved DRAM region to an arbitrary location in DRAM. (The BIOS vulnerability was also found by Rafal Wojtczuk and Alexander Tereshkin and published at the Black Hat conference in 2009.)
5. Exploit the BIOS bug and redirect the reserved region to a region that can be written by attack.
6. Debug the engine's firmware and hook an application that writes data to the host memory via DMA.
7. Inject rootkit to the DRAM region. The rootkit writes to host memory through DMA.

It should be pointed out that the attack is only possible on Bearlake MCH, released in 2007. The management engine on Bearlake MCH lacks integrity protection on the reserved region of the DRAM. This is one of the vulnerabilities exploited by the attack. Intel implemented the ICV check mechanism for the reserved DRAM region in the management engine released in 2008.

The attack takes advantage of two vulnerabilities. The other one is a buffer overflow in an older version of BIOS. Although the BIOS was patched soon after the issue was reported, BIOS downgrade was not disallowed. The lesson shows how firmware rollback prevention and integrity protection are vital to computer security.

However, the attack has some limitations:

1. It must hook an application that uses DMA. The researchers did not find a way to have the rootkit program DMA directly.
2. There is no way to perform DMA without redirecting memory remapping for BIOS. The remapping clears upon reboot.
3. Not all host memory is open to the embedded engine. For example, as mentioned earlier in this chapter, the VT-d and SMM memory cannot be accessed through the embedded engine's DMA.

References

1. Tereshkin, Alexander, and Rafal Wojtczuk, "Introducing Ring -3 Rootkits," Black Hat USA, July 29, 2009, Las Vegas, NV.
2. Wojtczuk, Rafal, and Alexander Tereshkin, "Attacking Intel® BIOS," Black Hat USA, July 29, 2009, Las Vegas, NV.
3. Intel Virtualization Technology, http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html, accessed on January 30, 2014.
4. Intel, "EFI System Management Mode Core Interface Spec (SMM CIS)," http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-smm-cis-v091.html, accessed on March 3, 2014.
5. National Institute of Standards and Technology, Common Vulnerability Scoring System (CVSS), http://nvd.nist.gov/cvss.cfm, accessed on December 12, 2013.
6. Request for comments 5246, "The Transport Layer Security (TLS) Protocol, Version 1.2," http://tools.ietf.org/html/rfc5246, access on December 12, 2013.
7. Brown, Rhonda, and Jackie Davenport, "Forensic Science: Advanced Investigations," Case Studies for Sasser Worm, Cengage Learning, 2012, pp. 414.
8. Huffman, D. A., "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the I.R.E., September 1952, pp. 1098–1102.
9. Pavlov, Igor, LZMA Software Development Kit, http://7-zip.org/sdk.html, accessed on December 12, 2013.
10. Hex-Rays, IDA disassembler, https://www.hex-rays.com/products/ida/, accessed on December 12, 2013.